

# CS1020: DATA STRUCTURES AND ALGORITHMS I

## Tutorial 11 – Hashing

(Week 13, starting 11 April 2016)

### 1. Hash Functions

A good hash function is essential for good hash table performance. A good hash function is easy/efficient to compute and will evenly distribute the possible keys. Comment on the flaw (if any) of the following hash functions. Assume the load factor  $\alpha$  = number of keys / table size = 0.7 for all the following cases:

- (a) The hash table has size 100. The keys are positive even integers. The hash function is  
$$h(\text{key}) = \text{key} \% 100$$
- (b) The hash table has size 49. The keys are positive integers. The hash function is  
$$h(\text{key}) = (\text{key} * 7) \% 49$$
- (c) The hash table has size 100. The keys are non-negative integers in the range of [0, 10000]. The hash function is  
$$h(\text{key}) = \lfloor \sqrt{\text{key}} \rfloor \% 100$$
- (d) The hash table has size 13. The keys are integers in the range of [0, 100]. The hash function is  
$$h(\text{key}) = \text{key} \% 13$$

Collision resolution uses double hashing with secondary hash function

$$h_2(\text{key}) = \text{key} \% 7$$
- (e) The hash table has size 1009. The keys are valid email addresses. The hash function is  
$$h(\text{key}) = (\text{sum of ASCII values of each of the last 10 characters}) \% 1009$$

See <http://www.asciitable.com> for ASCII values
- (f) The hash table has size 101. The keys are integers in the range of [0, 1000]. The hash function is  
$$h(\text{key}) = \lfloor \text{key} * \text{random} \rfloor \% 101, \text{ where } 0.0 \leq \text{random} \leq 1.0$$

### 2. Hashing and Collision Resolution

Consider a hash table with size  $m = 13$ . First, the following 11 keys are **inserted** into the table in the following sequence:

10    100    32    45    58    126    3    29    200    400    0

Next, the following 4 operations are performed in sequence, assuming that **lazy deletion** is used:

Delete(45), Delete(4), Find(58) and Find(45)

- (a) Show the contents of the hash table after each insert, delete and find operation. Use a simple hash function  $h(\text{key}) = \text{key} \% m$ , and **linear probing** to resolve collisions.
- (b) Repeat (a) by using the following hash function and **quadratic probing** to resolve collisions.

```
int hash(int key, int m) {  
    int sum = digitSum(key); //digitSum adds all digits of key,  
    return sum % m;          //e.g. key = 126, sum = 1+2+6 = 9.  
}
```

### 3. Binary Search on Sorted Array vs Lookup in Hash Table

An array named `values` contains 511 distinct integers in ascending order. The following code is used to determine if an integer `key` exists within `values`:

```
int leftIndex = 0;
int rightIndex = values.length - 1;
while (leftIndex <= rightIndex) {
    int middleIndex = (leftIndex + rightIndex)/2;
    if (values[middleIndex] == key) {
        return true;
    } else if (values[middleIndex] < key) {
        leftIndex = middleIndex + 1;
    } else {
        rightIndex = middleIndex - 1;
    }
}
return false;
```

**Restriction** for all parts: Search for each `key` independently. Do NOT reuse the result from one search in another. Do NOT search for two keys at once.

(a) Compute the total number of **key comparisons** necessary to locate all 511 values in the array using the above loop. A key comparison occurs whenever we evaluate one of:

- `if (values[middleIndex] == key) { ...`
- `} else if (values[middleIndex] < key) { ...`

(b) Now suppose that the 511 distinct integers are already stored in a hash table with size  $p$ , where  $p$  is a prime. The hash function is  $h(key) = key \% p$ . We use **separate chaining** to resolve collisions. We assume that all the 511 integers are uniformly distributed, i.e. there are either  $\left\lceil \frac{511}{p} \right\rceil$  or  $\left\lfloor \frac{511}{p} \right\rfloor$  nodes in each chain.

**Determine the minimal  $p$** , i.e. the smallest possible hash table, such that the total number of key comparisons necessary to access all 511 values in this hash table is smaller than or equal to the result in (a). Keep in mind that even though the first node in the chain is the value we are searching for, we still need a key comparison. Checking whether there is a node after the current does NOT count as a key comparison.

(c) One of the problems with *separate chaining* is that long chains slow down the search, because we can only compare keys sequentially. One suggestion is: instead of using linked list as a **chain**, use a **sorted array**.

Each slot in the hash table contains a reference to a sorted array. The arrays may have different sizes. When finding whether a key exists in the hash table, we can use the hash function and table size  $p$  in (b), but resolve collisions by inserting the colliding key into the appropriate sorted array.

As all arrays are sorted, we can now apply binary search to improve the efficiency. Suppose the 511 integers are already stored in the new hash table, does this improve searching over (a) and (b)? Also, is this design good, and why or why not?

Note: Question 4 will not be discussed during tutorial. You may discuss it on the IVLE forum.

#### 4. Finding Combinations

Given an **odd** integer  $k$ , and a list of  $n$  people, each having a name and amount of money, **find all pairs** of people who have \$ $k$  in total. For each pair found, print their (name, amount). You need not print the other permutation of the same pair of people. Every person has a unique name.

### Example

Given  $k = 9$  and the following list of 5 people:

(Alice, 4), (Bob, 3), (Charlie, 7), (Kenneth, 5), (Victor, 2)

The output should be:

```
(Alice, Kenneth)
(Charlie, Victor)
```

**Restriction:** Use a `java.util.HashMap` to solve the problem. You may also use other data structures along with the `HashMap` if really necessary.

```
class Person {
    private String _name;
    private int _amount;
    public Person(String name, int amount) {
        _name = name; _amount = amount;
    }
    public String getName() { return _name; }
    public int getAmount() { return _amount; }
}

public void findAllPairs(List<Person> people, int targetAmount) {}
```

- (a) Each person has a unique amount of money. Design an  $O(n)$  algorithm to solve the problem.
- (b) More than one person may have the same amount of money. Analyze the time complexity of your solution.

